International Journal of Information Security manuscript No.
(will be inserted by the editor)

J. G. Cederquist · R. Corin · M. A. C. Dekker
S. Etalle · J. I. den Hartog · G. Lenzini

The Audit Logic

Policy Compliance in Distributed Systems

Abstract We present a distributed framework where agents can share data along with usage policies. We use an expressive policy language including conditions, obligations and delegation. Our framework also supports the possibility to refine policies. Policies are not enforced a-priori. Instead policy compliance is checked using an a-posteriri auditing approach. Policy compliance is shown by a (logical) proof that the authority can systematically check for validity. Tools for automatically checking and generating proofs are also part of the framework.

Keywords Access Control · Audit · Policy · Privacy

1 Introduction

In many situations, there is a need to share data between potentially untrusted parties while ensuring the data is used according to given *policies*.

For example, Alice may need to send CompanyX her e-mail address, but at the same time she would like CompanyX not to disclose her email to anyone else. Alice could attach a *policy* (e.g. a P3P policy, see below) to her email address reflecting this desire but nothing can give Alice the guarantee that CompanyX will actually follow this policy.

J. G. Cederquist has been supported by NWO project "ACCOUNT". R. Corin and M. A. C. Dekker have been supported by IOP Generic Communication project "PAW". J. I. den Hartog has been supported by the EU project "INSPIRED". G. Lenzini has been supported by EU-ITEA project "Trust4All".

J. G. Cederquist · R. Corin · S. Etalle · J. I. den Hartog
Department of Computer Science, University of Twente, The Netherlands E-mail: cederquistj,corin,etalle,hartogji@cs.utwente.nl

M. A. C. Dekker
Security Group, TNO ICT, The Netherlands E-mail: m.a.c.dekker@telecom.tno.nl

G. Lenzini
Telematica Instituut, Enschede, The Netherlands E-mail: gabriele.lenzini@telin.nl

This situation is common not only when dealing with web services, but also in a collaborative environment such as a cross-organization cooperation (when an employee would like its memo to be used only within a certain project) or in the medical world (where the health record may be modified by various doctors, but should remain confidential, e.g. it should not be disclosed to insurance companies).

There are two main research streams addressing the problem of guaranteeing that information is actually used in accordance to policies: on one hand, there is a large body of literature on *access and usage control* [14,22,17,6], on the other hand we find *digital rights management* (DRM) [24, 8]. While the former assumes a trusted access control service restricting data access, the latter assumes a trusted device in charge of content rendering. Both settings need the trusted components to be available at the moment the request happens, to regulate the data access.

However, access control is unsuitable for the collaborative environments we are focusing on, as once an authorization has been granted there is no mechanism for controlling how the data is propagated further. DRM on the other hand is too rigid and restrictive: all users would have to use the same special-purpose software and hardware. Also merging two documents and retransmitting the result to another user is beyond the possibilities of DRM systems. Indeed, assuming the availability of trusted access control services to restrict data access at the moment the request happen can be sometimes overly restrictive and expensive.

For instance, in the case of protection of private data, there exist a number of frameworks that have been developed, but none of them gives the guarantees we are looking for: the Platform for Privacy Preferences Project (P3P) [23] is a standardized, XML-based, policy specification languages that can be used to specify the organization's privacy practices in a way that can be parsed and used by policy-checking agents on a user's behalf. However, with P3P the user has no guarantee that an organization follows the policies it has claimed, because the user has no control over the organization's actions. As another example, the IBM Enterprise Privacy Authorization Language (EPAL) [4] is an XML-based privacy policy specification language designed for organi-

zations to specify and implement internal privacy policies. EPAL policies can be used throughout an organization and its business partners to ensure compliance with their underlying policies. Also, employees can use EPAL as an "advisor", to be sure of not violating any policy before doing an action; However, is is not possible to check if an employee has indeed violated a policy.

In this paper we develop the fundamental basis for a different, more flexible approach, where no active enforcement is performed. In our system access control is not enforced *a-priori* but is checked *a-posteriori*. Misuse is not prevented but deterred: One or more *auditing authorities* have a mandate of checking whether the data was used in compliance with the policies. Hence users should be *auditable* and sufficient *audit trails* should be available to the auditors. This fits well with e.g. hospitals or companies, where users can be held accountable for their actions and audit trails are often already part of the (security) requirements. It may be hard to realize these requirements in other settings, such as large open networks, or peer to peer (P2P) networks. (However, this trend may be changing, as for instance nowadays EU law demands that ISP's keep IP traffic records of all their users.)

Going back to the example, in our system, after CompanyX receives Alice's e-mail address, CompanyX could violate the policy, for instance by sending Alice's e-mail to BadCompanyY and including a "free-to-be-spammed" policy. However, CompanyX may later be *audited* by an authority that requests a convincing proof of CompanyX's permission to disclose Alice's e-mail address.

In this paper, we develop a formal framework which describes the fundamental basis for a system implementing a-posteriori usage control. In particular, our contribution is threefold:

- We develop an expressive logic-based policy language to describe *sticky* policies, i.e. policies that follow the data when it is moved across security domains. The language enables to specify both conditions and obligations.
- We define a derivation system for the language supporting the concept of accountability, a notion specifically tailored to a-posteriori access control.
- We develop and implement two complementary, pivotal elements of our architecture: A *proof checker* that allows an auditing authority to check justification proofs provided by agents, and a corresponding *proof finder* which allows agents to generate valid justification proofs.

Figure 1 shows an example execution in the framework: In the first step (I), agent a provides a policy ϕ to agent b which b records in his log (II). Next (III) agent b reads document d which is stored in the company database. At a later point the auditing authority, which is checking access to privacy sensitive files, finds the access of b (IV) and requests b to justify this access (V). In response, b shows that the access was allowed according to the policy ϕ which was provided by a. The auditor, initially unaware of a's involvement, can now (VI) audit a for having provided the policy ϕ to b.

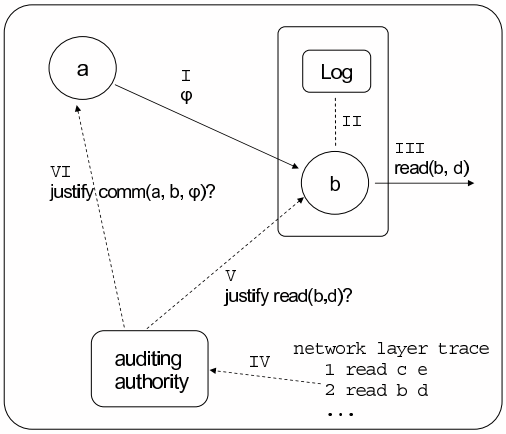

Fig. 1 Sample deployment depicting actions, the logging and interaction with an auditor.

This paper builds on preliminary work [7,9]. In Corin et al. [9] we outline our framework, and develop the core notions of data and agent accountability. In Cederquist et al. [7] we extend the policy language to allow variables and quantifiers. This allows us to define a fundamental rule that gives the ability to refine policies. Agents can create (by refinement) new policies from existing ones, before passing them to other agents. (In contrast, in [9] the only policies allowed are those that are explicitly stated by the data owner.) Also, [7] describes our system more precisely by introducing three functions, namely the *observability*, *conclusions* and *proof obligation* functions; it also provides a prototype implementation of the proof checker in the Twelf logical framework [19], which allows to model agents providing proofs, and authorities checking them.

In this paper we provide a revision of [7,9], with detailed examples, extensions and further explanations. In addition, as a novel significant contribution, we further develop our proof system, and prove the key result of cut-elimination. This allows us to implement an efficient proof finder that complements our proof checker, and that agents may use to find valid justification proofs.

The rest of this paper Next, in Section 2 we describe our policy language, along with a description of the proof system. In Section 3 we present the system model and the notion of accountability. Section 4 describes the proof finder and proof checker. Section 5 discusses the main advantages and applicability of our approach. We discuss related work in Section 6. Section 7 provides conclusions and future work. Appendix A provides a proof of the cut-elimination property for our proof system.

2 Actions and Policies

Our setup consists of a group of agents executing different actions. The permission to execute an action is expressed

by a policy constructed using a special logic. This section introduces the components of the architecture, presents the language we use to express usage policies, and explains how to reason about policies at audit time.

2.1 Basic Sets

We assume the presence of a set $\mathcal{AG} = \{Alice, Bob, \ldots\}$ of *agents* and a set $\mathcal{DO}$ of *data objects*. We let a, b, c range over $\mathcal{AG}$, and d over $\mathcal{DO}$.

2.2 Policies: Permissions, Facts and Actions

Policies are used to express *permissions* granted to agents, (*e.g.,* the permission to read a specific piece of data) and *facts* are used to express agents properties (*e.g.,* the fact that an agent is a member of a certain club). We model atomic permissions and facts as n-ary predicates, $p(s_1, \ldots, s_n)$.

Example 1 Examples are $\texttt{mayRead}(a, d)$, which expresses that agent a has permission to read data d and $\texttt{partner}(a, b)$ indicating (the fact) that agent a and b are partners.

Permissions may be guarded by some requirements as in "Alice may read the data if she is a partner of Bob", or as in "Alice may read the data if she pays Bob a one time fee of $10". These requirements can be *conditions i.e.,* policies like "Alice is partner of Bob", or *obligations i.e.,* requirements over an action "Alice pays Bob a one time fee of $10". (See also [21] for a discussion on conditions and obligations.)

We now introduce our formal policy language:

Definition 1 (Policy Language) The set $\mathcal{PO}$ of *policies*, ranged over by ϕ and ψ, is defined by the following grammar:

$$
\begin{aligned}
\phi ::= {} & p(s_1, ..., s_n) \\
\mid {} & a \ \texttt{owns} \ d \\
\mid {} & a \ \texttt{says} \ \phi \ \texttt{to} \ b \\
\mid {} & \top \mid \phi \wedge \phi \mid \forall x.\phi \mid \phi \rightarrow \phi \mid \xi \rightarrow \phi \\
\xi ::= {} & !act \mid ?act
\end{aligned}
$$

There are two kinds of basic policies. First, predicates of the form $p(s_1, ..., s_n)$, where the s_i are agents or data objects (possibly variables), express a basic permission or a fact. Second, the policy $a \ \texttt{owns} \ d$, indicates that a is the owner of data object d. A data owner is allowed to create usage policies related to that data. The other policies are *compound*, as they contain sub-policies. The policy $a \ \texttt{says} \ \phi \ \texttt{to} \ b$ indicates that agent a is allowed to communicate policy ϕ to agent b. Note that, differently from [2], this policy contains a target agent. This feature allows us to provide a precise way of communicating policies to certain agents. The policy $a \ \texttt{says} \ \phi \ \texttt{to} \ b$ has a different meaning for source agent a than target agent b; for agent a it represents the permission to send ϕ to b, while for b it represents the permission to use policy ϕ. The trivial policy $\top$ does not give any permissions. The conjunction $\wedge$ and the universal quantification $\forall$ have their usual meaning. On the other hand implication $\rightarrow$, can be used in two ways. The first, $\phi \rightarrow \psi$, has a policy ϕ as a *condition*, stating that the agent first needs to satisfy ϕ before obtaining the permission described in ψ. The second, $\xi \rightarrow \phi$, is used to express obligations. In this case, the requirement ξ contains an action act that the agent has to perform before using ϕ. The annotations ! and ? indicate whether the agent needs to do this action every time it uses ϕ, in $\xi \rightarrow \phi$, or if it only needs to do the action once. We will refer to an obligation ξ as *use-once* or *use-many* obligation, respectively. This will be discussed in more detail in Section 3.3.

For simplicity, in our policy language, policies can not be formed using disjunction or existential quantification.

Actions are chosen from a set $\mathcal{AC}$, that contains at least $\texttt{comm}(a \Rightarrow b, \phi)$, expressing a communication of a policy ϕ from agent a to b, and $\texttt{creates}(a, d)$, expressing that the creator of data d is a. Our system also supports the addition of user-defined actions.

We use the notation $data(\phi)$ for the set of data objects and data variables that occur in the policy ϕ. For instance, we have $data(\texttt{read}(b, d)) = \{d\}$.

Example 2 (Policies) The scenario considered in this example assumes a system containing a video studio, a rating service and a customer. The video studio manages a web site where clips and trailers are available to be downloaded and played. The material is subject to a rating system with three classes: the first class is suitable for all ages, the second is suitable for people over 13 years old, the last one is suitable for people over 17. We express that a content d is subjected to the rating "for all", " over 13", or "over 17" with the following predicates:

$$\texttt{ratedAll}(d), \qquad \texttt{ratedPG13}(d), \qquad \texttt{ratedNC17}(d)$$

The material owned by the studio is distributed together with the following set of rate policies:

$$
\begin{aligned}
\texttt{ratedAll}(d) &\rightarrow \phi \\
\texttt{ratedPG13}(d) \wedge \texttt{ageover13}(b) &\rightarrow \phi \\
\texttt{ratedNC17}(d) \wedge \texttt{ageover17}(b) &\rightarrow \phi
\end{aligned}
$$

where the usage policy ϕ can be:

- A "subscription service": The subscriber b is allowed to play any content owned by the Studio:
  $$\forall x.(Studio \ \texttt{owns} \ x \rightarrow \texttt{mayPlay}(b, x))$$
- A "buy content" policy which allows a subscriber a to play the content d as often as he wants after having payed 10$ once.
  $$?\texttt{pay}(a, 10\$) \rightarrow \texttt{mayPlay}(a, d)$$
- A "pay per view" policy which allows a to play the content d once after having payed 1$.
  $$!\texttt{pay}(b, 1\$) \rightarrow \texttt{mayPlay}(b, d)$$

$$\textsc{refine}\ \dfrac{\dfrac{\dfrac{\dfrac{[\texttt{print}(b,d)]}{\texttt{rel}(d,y)\rightarrow\texttt{print}(b,d)}\,{\scriptstyle\rightarrow_{\mathrm{I}}\,a}}{\forall x.\texttt{rel}(d,x)\rightarrow\texttt{print}(b,d)}\,{\scriptstyle\forall_{\mathrm{I}}\,a}}{\texttt{print}(b,d)\rightarrow\big(\forall x.\texttt{rel}(d,x)\rightarrow\texttt{print}(b,d)\big)}\,{\scriptstyle\rightarrow_{\mathrm{I}}\,a} \qquad \textsc{owns}-\textsc{e}\ \dfrac{\textsc{concl}\ \dfrac{\texttt{creates}(a,d)}{a\ \texttt{owns}\ d}\,{\scriptstyle a}}{a\ \texttt{says print}(b,d)\ \texttt{to}\ b}\,{\scriptstyle a}}{a\ \texttt{says}\ \big(\forall x.\texttt{rel}(d,x)\rightarrow\texttt{print}(b,d)\big)\ \texttt{to}\ b}\,{\scriptstyle a}$$

Fig. 2 Derivation tree that shows the use of the rule REFINE and DER_POL. Agent a derives the policy $a\ \texttt{says}\ \varphi\ \texttt{to}\ b$ that is the policy that allows her to give to another agent b the policy φ, where $\varphi = \big(\forall x.\texttt{rel}(d,x)\rightarrow\texttt{print}(b,d)\big)$.

2.3 Observability, Proof Obligation and Conclusion

Here we describe the meaning of different actions in terms of a) the set of agents that can observe them, b) the policy an agent needs to justify in order to perform an action, and c) the conclusion an agent can draw by observing an action. These three properties of actions, which play a fundamental role in our policy system, are described by the following functions:

- The *observability* function: $obs : \mathcal{AC} \rightarrow 2^{\mathcal{AG}}$, where $2^{\mathcal{AG}}$ is the powerset of $\mathcal{AG}$, describing which agents can observe which actions.
- The *proof obligation* function: $pro : (\mathcal{AC} \times \mathcal{AG}) \rightarrow \mathcal{PO}$ describes which policy an agent needs to satisfy in order to justify the execution of an action.
- The *conclusion derivation* function: $concl : (\mathcal{AC}\times\mathcal{AG}) \rightarrow \mathcal{PO}$, describes what policy can an agent deduce after observing an action.

For the default actions, $\texttt{creates}(a,d)$ and $\texttt{comm}(a \Rightarrow b, \phi)$, we have:

$$obs(\texttt{creates}(a,d)) \ni \{a\} \tag{1}$$
$$obs(\texttt{comm}(a \Rightarrow b,\phi)) \supseteq \{a,b\} \tag{2}$$
$$pro(\texttt{creates}(a,d),b) = \top \tag{3}$$
$$pro(\texttt{comm}(a \Rightarrow b,\phi),a) = a\ \texttt{says}\ \phi\ \texttt{to}\ b \tag{4}$$
$$pro(\texttt{comm}(a \Rightarrow b,\phi),c) = \top \quad (a \neq c) \tag{5}$$
$$concl(\texttt{creates}(a,d),a) = a\ \texttt{owns}\ d \tag{6}$$
$$concl(\texttt{creates}(a,d),b) = \top \quad (b \neq a) \tag{7}$$
$$concl(\texttt{comm}(a \Rightarrow b,\phi),b) = a\ \texttt{says}\ \phi\ \texttt{to}\ b \tag{8}$$
$$concl(\texttt{comm}(a \Rightarrow b,\phi),c) = \top \quad (c \neq b) \tag{9}$$

This can be explained intuitively as follows: (1) a creation action is observable by (at least) the agent who performed the action. (2) a communication is observable by (at least) the source and target agent. (3) agents do not need permissions for creating data. (4) in a communication, the source agent needs a permission. (5) other agents do not. (5) an agent who creates data can conclude that it is the owner of the data. (6) other agents cannot conclude anything from a creation action. (7)) the target agent in a communication can conclude the corresponding says policy. (8) other agents cannot conclude anything from a communication.

Remark 1 The action $\texttt{comm}(a \Rightarrow b,\phi)$ models point-to-point communication. We can model broadcasting, by introducing an action $\texttt{bcast}(a,\phi)$, and setting:

$$obs(\texttt{bcast}(a,\phi)) = \mathcal{AG}$$
$$pro(\texttt{bcast}(a,\phi),x) = \begin{cases} \forall y.a\ \texttt{says}\ \phi\ \texttt{to}\ b & \text{, if } (x=a) \\ \top & \text{, otherwise} \end{cases}$$
$$concl(\texttt{bcast}(a,\phi),x) = \forall y.a\ \texttt{says}\ \phi\ \texttt{to}\ b,$$

Here, every agent can observe the action $\texttt{bcast}(a,\phi)$ and conclude that a has broadcast ϕ *i.e.,* said ϕ to everybody. Notice that only a needs to justify this action.

From an operational point of view we distinguish between actions and *instantiations* of actions. Moreover, we want to distinguish between different instances of the same action, which is done by labeling each instance with a unique identifier id, as in $\texttt{creates}_{id}(a,d)$. Formally this gives a set $\mathcal{AC}^* \in \mathbb{N} \rightarrow \mathcal{AC}$ of "action instantiations". While the observability and proof obligation functions depend on instances of actions (*i.e.,* an identifier), the conclusion derivation function is purely syntactical.

Example 3 We gather the basic components of the framework needed to describe the scenario introduced in Example 2. We have the following sets of agents:

$$\mathcal{AG} = \{Alice, Studio, Rating, Bank\}$$

Here $Alice$ denotes the customer, $Studio$ denotes the video studio, and $Rating$ denotes the rating service. Morover, we have included an additinal entity $Bank$, the bank where the custor can pay the fee required by an usage police.

The set of data $\mathcal{D}$ is composed by two elements, $clip$ and $trailer$; they model the clip and the trailer, respectively, that are the targets of this example. The set of possible actions is as follows:

$$\mathcal{AC} = \{\texttt{creates}(b,d), \texttt{comm}(b \Rightarrow d,\varphi)\}$$
$$\{\texttt{pay}(b,n\$), \texttt{play}(b,d)\}$$

The first two actions are those included for default in the set of actions (see Section 2.3) where φ ranges over the policies we have described in the previous section; the last two model the action of playing, by entity b, of a material d, and the action of paying, performed by b, of a certain amount n of dollars, respectively; their meaning is moreover described in terms of observatibility, proof obligation and conclusion

derivation as follows:

$$\begin{aligned}
obs(\texttt{pay}(Alice, n\$)) &= \{Alice, Bank\} \\
obs(\texttt{play}(Alice, trailer)) &= \{Alice\} \\
obs(\texttt{play}(Alice, clip)) &= \{Alice\} \\
pro(\texttt{pay}(b, n\$), c) &= \top \\
pro(\texttt{play}(b, d), b) &= \texttt{mayPlay}(b, d) \\
pro(\texttt{play}(b, d), c) &= \top \qquad \text{if } c \neq b \\
concl(\texttt{pay}(b, n\$)) &= \top \\
concl(\texttt{play}(b, d), c) &= \top
\end{aligned}$$

2.4 Deriving Policies

This section describes how an agent can derive policies, when it needs to demonstrate that it had permissions to carry out the actions it did. This is done by using a *derivation system* or *proof system*. In this section, the proof system is described informally, while in the next section we give the formal definition of it. The derivation system contains the standard predicate logic rules for introduction and elimination of conjunction, implication and universal quantification, together with the following rules:

$$\textsc{says}-\textsc{e}\ \frac{b\ \texttt{says}\ \phi\ \texttt{to}\ a}{\phi}\ a$$

$$\textsc{refine}\ \frac{taut(\wedge_{i\leq n}\phi_i \rightarrow \psi) \quad a\ \texttt{says}\ \phi_i\ \texttt{to}\ b\ \ (\forall i \leq n)}{a\ \texttt{says}\ \psi\ \texttt{to}\ b}\ a$$

$$\textsc{concl}\ \frac{act}{concl(act, a)}\ a$$

$$\textsc{owns}-\textsc{e}\ \frac{data(\phi) \subseteq \{d_1, \ldots, d_n\} \quad a\ \texttt{owns}\ d_i\ \ (\forall i \leq n)}{\phi}\ a$$

Note that each step in a derivation carries the name a of the agent that is doing the reasoning.

The rule (SAYS−E) models the delegation of a policy. If agent a can derive $b\,\texttt{says}\,\phi\,\texttt{to}\,a$ then a can assume ϕ to hold. Agent a may use ϕ without further requirement and it is b's responsibility to show that it had permission to give ϕ to a, see Section 3.3 on accountability. Using the rule (REFINE) an agent can refine the policies it can delegate. The policy ψ is a refinement of ϕ if $\phi \rightarrow \psi$ is a tautology (for all agents, at all times). Rule (CONCL) links an action act with its conclusion, given by the conclusion derivation function $concl$. For instance, from observing action $\texttt{comm}(a \Rightarrow \phi, b)$, agent b derives $a\ \texttt{says}\ \phi\ \texttt{to}\ b$.

As mentioned already, we designed the logic in such a way that the owner of some data d decides who is allowed to do which actions on d. In other words, the owner of some data d is allowed to derive any policy for d. Rule (OWNS−E) achieves this goal; it allows the creation of any policy for data which the agent owns. Non-owners may refine existing policies (*e.g.,* policies they received), but may not create new policies from scratch.

Example 4 (Policy Refinement) To illustrate the usage of the rules (REFINE) and (OWNS−E), suppose that $\texttt{rel}(d, d')$ expresses that there is a review d of a new product and d' is the press release announcing this product. Agent a creates a new object d and wants to give to agent b the policy

$$\forall x.\texttt{rel}(d, x) \rightarrow \texttt{print}(b, d).$$

This policy gives b the permission to print the document as soon as a related object exists. Agent a can build the policy allowing her to give this policy to b as shown in Figure 2.

2.5 Proof System

We now present the proof system that was discussed informally in the previous section.

The audit logic is formalized as an intuitionistic logic using sequent calculus. We believe that in our framework where the authority during auditing may inquire several agents, the use of constructive proofs makes it easier for the authority keep track of the chains of responsibilities. A proof by contradiction of the policy "there exists an agent who told me that I am allowed to ..." for instance would not tell the authority who gave the permission.

There are two reasons for using sequent calculus for the formalization: First, the sequent calculus uses a notation that is explicit about which assumptions are used at which step in the proof. This is convenient because the agents may use different assumptions. Below we use the (sequent) notation $\Gamma \vdash_a \phi$ to indicate that agent a can prove ϕ by using the assumptions in Γ. The second reason is more practical: proof search in sequent calculi can be done almost entirely by a simple backtracking search. This has allowed us to implement a proof finder in Prolog in a straightforward way (see Section 4.3).

The proof system is shown in Figure 3. As earlier, ϕ and ψ denote policies, while α denotes an action. Sequents have the form $\Gamma_1; \Gamma_2; \Delta \vdash_a \phi$, where a is the agent doing the reasoning, and Γ_1, Γ_2 and Δ are three different contexts. The context Γ_1 is a list of policies. The context Γ_2 is a list of actions from the agent's log, which are used to derive conclusions using the conclusion derivation function $concl$, or as use-once obligations. Finally, the context Δ is a linear[1] context, which is used to model use-once obligations $obligs$. The empty context is denoted ϵ. To keep the notation as simple as possible, when a context is the same in the conclusion as in the premises, it is left out from the rule. Thus, instead of writing

$$\frac{\Gamma_1; \Gamma_2; \Delta \vdash_a \phi \quad \Gamma_1; \Gamma_2; \Delta' \vdash_a \psi}{\Gamma_1; \Gamma_2; \Delta, \Delta' \vdash_a (\phi \wedge \psi)} \wedge\text{R}$$

we write

$$\frac{;\,;\Delta \vdash_a \phi \quad ;\,;\Delta' \vdash_a \psi}{;\,;\Delta, \Delta' \vdash_a (\phi \wedge \psi)} \wedge\text{R}.$$

[1] In linear logic assumptions are used exactly once, while our logic allows weakening. It would be more exact to say that Δ is an *affine* context.

$$\frac{}{\Gamma_1;\Gamma_2;\Delta \vdash_a \top}\top\mathrm{R} \qquad \frac{}{\Gamma_1,\phi;\Gamma_2;\Delta \vdash_a \phi}\mathrm{I}$$

$$\frac{\Gamma_1,\phi_1 \vdash_a \psi}{\Gamma_1,(\phi_1 \wedge \phi_2);;\vdash_a \psi}\wedge\mathrm{L}_1 \qquad \frac{\Gamma_1,\phi_2;;\vdash_a \psi}{\Gamma_1,(\phi_1 \wedge \phi_2);;\vdash_a \psi}\wedge\mathrm{L}_2$$

$$\frac{\Gamma_1;;\Delta \vdash_a \phi_1 \quad \Gamma_1,\phi_2;;\Delta' \vdash_a \psi}{\Gamma_1,(\phi_1 \to \phi_2);;\Delta,\Delta' \vdash_a \psi}\to\mathrm{L}$$

$$\frac{\Gamma_1,\phi(x);;\vdash_a \psi}{\Gamma_1,\forall y.\,\phi(y);;\vdash_a \psi}\forall\mathrm{L}$$

$$\frac{\Gamma_1,\phi;;\Delta \vdash_a \psi}{\Gamma_1,(!\alpha \to \phi);;\Delta,\alpha \vdash_a \psi}!\to\mathrm{L}$$

$$\frac{\Gamma_1,\phi;\Gamma_2;\vdash_a \psi}{\Gamma_1,(?\alpha \to \phi);\Gamma_2,\alpha;\vdash_a \psi}?\to\mathrm{L}$$

$$\frac{\Gamma_1,\phi,\phi;;\vdash_a \psi}{\Gamma_1,\phi;;\vdash_a \psi}\text{C-L}_1$$

$$\frac{\Gamma_1,\phi;;\vdash_a \psi}{\Gamma_1,\mathsf{says}(b,\phi,a);;\vdash_a \psi}\text{says-L}$$

$$\frac{data(\phi) \subseteq \{d_1,\ldots,d_n\}}{\Gamma_1,\mathsf{owns}(a,d_1),\ldots,\mathsf{owns}(a,d_n);\Gamma_2;\Delta \vdash_a \phi}\text{owns-L}$$

$$\frac{\Gamma_1;;\Delta \vdash_a \phi \quad \Gamma_1,\phi;;\Delta' \vdash_a \psi}{\Gamma_1;;\Delta,\Delta' \vdash_a \psi}\text{cut}$$

$$\frac{;;\Delta \vdash_a \phi \quad ;;\Delta' \vdash_a \psi}{;;\Delta,\Delta' \vdash_a (\phi \wedge \psi)}\wedge\mathrm{R}$$

$$\frac{\Gamma_1,\phi;;\vdash_a \psi}{\Gamma_1;;\vdash_a (\phi \to \psi)}\to\mathrm{R}$$

$$\frac{;;\vdash_a \phi(x)}{;;\vdash_a \forall y.\,\phi(y)}\forall\mathrm{R}$$

$$\frac{;;\Delta,\alpha \vdash_a \phi}{;;\Delta \vdash_a (!\alpha \to \phi)}!\to\mathrm{R}$$

$$\frac{;\Gamma_2,\alpha;\vdash_a \phi}{;\Gamma_2;\vdash_a (?\alpha \to \phi)}?\to\mathrm{R}$$

$$\frac{;\Gamma_2,\alpha,\alpha;\vdash_a \psi}{;\Gamma_2,\alpha;\vdash_a \psi}\text{C-L}_2$$

$$\frac{\Gamma_1;\epsilon;\epsilon \vdash_a \psi}{\Gamma_1',\mathsf{says}(b,\Gamma_1,c);\Gamma_2;\Delta \vdash_a \mathsf{says}(b,\mathsf{says}(b,\psi,c))}\text{refine}$$

$$\frac{\Gamma_1,\mathit{concl}(\alpha,a);\Gamma_2;\vdash_a \psi}{\Gamma_1;\Gamma_2,\alpha;\vdash_a \psi}\text{concl}$$

Fig. 3 The proof system used in the tools.

The first ten rules in the proof system are standard rules for $\top$, initialization, cut and, left and right rules for conjunction, implication and universal quantification. The next four rules are the implication left and right rules for the use-one and use-many obligations. There are the two contraction rules (C-L_1 and C-L_2) for the two non-linear contexts. The final four rules, says-L, refine, owns-L and concl, correspond to the rules SAY−E, REFINE, OWNS−E and CONCL, introduced in Section 2.5. In the conclusion of the refine rule, the formula $\mathtt{says}(b,\Gamma_1,c)$ is used as an abbreviation for list of policies $\mathtt{says}(b,\phi,c)$ where ϕ in Γ_1. In addition to the rules shown in Figure 3, there are also permutation rules, one for each context.

We now discuss the refine rule in greater detail. The action contexts, in the premise are empty, because (in our framework) local reasoning should not have any influence of what can be a refinement of a policy. More precisely, logged actions are local to agents, and so are the conclusions of these actions. (Cf. the tautology requirement in the REFINE rule, in Section 2.5.) If non-empty action contexts were allowed, then an agent may have concluded a certain policy from an action and then communicated that policy to an other agent.

In the audit logic, as mentioned in Section 2.5, if an agent can derive a certain policy, it does not necessarily mean that it can communicate that policy to other agent. The presence of the contexts Γ',Γ_2 and Δ is to allow for a proof of *weakening*[2], which is a derived rule in our logic.

[2] Weakening says that, if a certain property ϕ can be derived from the assumptions Γ, then ϕ can also be derived from Γ,ψ.

We return to the owns-L rule for a discussion about implementation issues in section 4.3.

3 The Model

We now introduce a model for our system, combining the different components of the previous sections. In our system, agents can execute and log actions. In addition to agents, there exist an authority which may audit agents requiring justification for (some of) the agents actions.

3.1 Logged Actions: The System State

Whenever an agent executes or observes an action, it can also choose to simultaneously *log* this action. Logged actions constitute evidences that can be used to demonstrate that an agent was allowed to perform a subsequent action, and are used during accountability auditing, in Section 3.3.

Definition 2 A *logged action* is a triple $lac = \langle act, \Gamma, \Delta\rangle$ consisting of an action $act \in \mathcal{AC}$, a set of facts $\Gamma \subseteq \mathcal{PO}$ (the conditions), and a set of actions $\Delta \subset \mathcal{AC}$ (the 'use-once obligations'). We use $\mathcal{AC}^*$ to denote the space of all logged actions.

When logging an action, an agent can include conditions, i.e. facts about the current situation that the environment certifies to be valid at the moment of execution of the action,

for example the current time. We do not model the environment explicitly but instead assume that the agent obtains a secure "package" of signed facts from this environment, represented by Γ. As an example, one can think of the driver's license of Alice being checked to certify that she is over 21. As an aside note that, to deal more efficiently with facts that remain true all the time, one could also have a set of 'global facts' which then do not have to be included in each logged action.

To satisfy the obligations in a policy in the list Δ an agent may also refer to other actions. This list refers to other actions the agent did or promises to do. We abstract away from the details of expressing promises, and instead assume we have a way to check if actions have *expired*. The agent has to perform and log the action before it expires. For example, the agent may promise to pay within a day. Then a payment action needs to be done (and logged) within a day of logging this obligation. (See also Section 3.3.)

Example 5 Suppose that we have an action $\texttt{drink}(x, y)$ and a corresponding permission $\texttt{mayDrink}(x, y)$, with

$$\begin{aligned} &\mathit{concl}(\texttt{drink}(x, y), z) = \top \\ &\mathit{pro}(\texttt{drink}(x, y), x) = \texttt{mayDrink}(x, y) \\ &\mathit{pro}(\texttt{drink}(x, y), z) = \top \quad (z \neq x) \end{aligned}$$

We also introduce an action $\texttt{pay}(x, y)$, with conclusion and obligation $\top$.

If a pays for and then drinks a beer, it can log these actions as follows: The payment is recorded in the logged action $lac_{\texttt{pay}}$ given by:

$$lac_{\texttt{pay}} = \langle \texttt{pay}_0(a, 10\$), \emptyset, \emptyset \rangle$$

Then, when drinking the beer paid for in the previous action, a logs $lac_{\texttt{drink}}$ given by:

$$\begin{aligned} lac_{\texttt{drink}} = \langle &\texttt{drink}_1(a, \mathit{beer}), \\ &\{\texttt{age21}(a), \texttt{alc}(\mathit{beer})\}, \\ &\{\texttt{pay}_0(a, 10\$)\}\rangle \end{aligned}$$

The *log of an agent* a is a finite sequence of logged actions. Note that it does not need to be a who performed the actions, but of course a has to observe an action to be able to log it. We say that agent a logs action act when $\langle act, \Gamma, \Delta\rangle$ is appended to the log of a for some list of conditions Γ and some list of obligations Δ. The log of an agent represents the (usable) information an agent has about the system, in other words, the logs of all agents constitute the system state.

Definition 3 A system state S is a collection of logs of the different agents, i.e. a mapping from agents to lists of logged actions $S : \mathcal{AG} \to \mathcal{AC}^*$. We denote by $\mathcal{S}$ the collection of all states.

Agents cannot just log anything; besides the assumptions that actions are only logged when they happen and that the conditions logged are certified by the environment we also assume some basic consistency properties of the logs:

- An agent logs the same action at most once, i.e. there cannot be two different logged actions $\langle act_{id}, \Gamma, \Delta\rangle$ and $\langle act_{id}, \Gamma', \Delta'\rangle$ in the log for the same action act_{id}.
- An action can only be used one time as a use-once obligation, i.e. an action act_{id} may not occur in the obligations Δ of two different logged actions in the log.
- An agent cannot log an expired action.

The consistency of the log does not have to be checked at time of logging, it is sufficient to check it at time of auditing.

3.2 Executing Actions in the System

The system model is defined as a labeled transition system:

Definition 4 A system is a tuple: $\langle \mathcal{S}, S_0, \mathcal{L}, \rightarrow\rangle$, where $\mathcal{S}$ the set of all states as introduced in Definition 3, $S_0 \in \mathcal{S}$ is the initial state in which all logs are empty, $\mathcal{L} = \mathcal{AC} \times \mathcal{P}(\mathcal{AG})$ is the transition labels consisting of an action and a set of agents that log that action, and $\rightarrow \subseteq \mathcal{S} \times LAB \times \mathcal{S}$ is the transition relation. We use the notation $S \xrightarrow{act,L} S'$ for $(S, (act, L), S') \in \rightarrow$.

A transition models an action happening in the system and being logged by some agents observing the action. Thus we have $S \xrightarrow{act,L} S'$ when L is a subset of $obs(act)$, $S'(A) = S(A)$ for $A \notin L$ and $S'(A) = S(A).\overline{act}$ for $A \in L$ where $\overline{act}$ is a log of action act by agent A. In other words, S' is the same as S except that act has been logged by the agents in L.

An *execution* of the system in the definition above consists of a sequence of transitions $S_0 \xrightarrow{act_1,L_1} \ldots \xrightarrow{act_n,L_n} S_n$, starting with the (empty) initial state S_0. The *execution trace* for this execution is $act_1 \ldots, act_n$. In a state S the log $S(a)$ of an agent a can also be seen as a trace of actions (by ignoring the conditions and obligations logged with the actions). As a's log is initially empty and a can only log actions that actually do occur, the log is a sub-trace of the execution trace, i.e. we have $S_n(a) \preceq tr$, where $\preceq$ denotes the sub-trace relation ($tr_1 \preceq tr_2$ iff tr_1 can be obtained from tr_2 by leaving out actions but maintaining the order of the remaining actions).

Example 6 (An Execution Trace) First we describe some events, then we give the execution trace in detail. First the Studio enlists the Rating Service to rate his content, and on the release date it also provides the permission to publish the ratings. The customer, Alice, download and watches content produced by Studio; when necessary she also performs the actions required by the policies that are attached to the materials she gets from the Studio. The execution trace is as follows.

- the Studio creates the content $clip$ and a $trailer$, and logs these actions.

$$\begin{aligned} &S_0 = \emptyset \\ &act_1 = \texttt{creates}(\mathit{Studio}, \mathit{clip}) \\ &act_2 = \texttt{creates}(\mathit{Studio}, \mathit{trailer}) \end{aligned}$$

As the Studio does not need to satisfy any conditions or obligation for creating the content, the creation is logged as $\langle act_i, \emptyset, \emptyset \rangle$ $(i = 1, 2)$. We use the notation $S \cdot a \mapsto \langle \_, \_, \_ \rangle$ to indicate that S is extended by appending action $\langle \_, \_, \_ \rangle$ to the log of agent a.

$$S_1 = S_0 \cdot \mathit{Studio} \mapsto \langle act_1, \emptyset, \emptyset \rangle$$
$$S_2 = S_1 \cdot \mathit{Studio} \mapsto \langle act_2, \emptyset, \emptyset \rangle$$

– The Studio enlists the rating service to rate this content. On the release date, the Studio provides permission to publish the ratings, with the following policy ϕ_{cr} (standing for "can rate"):

$$\begin{aligned}\phi_{cr}(a, d) := \forall x.\ & a \ \texttt{says}\ (\texttt{ratedAll}(d))\ \texttt{to}\ x \\ & \wedge\ a \ \texttt{says}\ (\texttt{ratedPG13}(d))\ \texttt{to}\ x \\ & \wedge\ a \ \texttt{says}\ (\texttt{ratedNC17}(d))\ \texttt{to}\ x\end{aligned}$$

The action of communicating the policy is performed by Studio, and logged by Rating:

$$act_3 = \texttt{comm}(\mathit{Studio} \Rightarrow \mathit{Rating}, \phi^1_{cr} \wedge \phi^2_{cr}),$$

where

$$\phi^1_{cr} = \phi_{cr}(\mathit{Rating}, \mathit{clip})$$
$$\phi^2_{cr} = \phi_{cr}(\mathit{Rating}, \mathit{trailer})$$

After the logging the state is updated as follows:

$$S_3 = S_2 \cdot \mathit{Rating} \mapsto \langle act_3, \emptyset, \emptyset \rangle$$

– Visiting the Studio's web-site Alice finds the trailer which the Studio provides for free. Trailers, by the way, are subjected to the following rate policies

$$\phi_{all}(a, d) := \texttt{ratedAll}(d)$$
$$\phi_{age13}(a, d) := \texttt{ratedPG13}(d) \wedge \texttt{ageover13}(a)$$
$$\phi_{age17}(a, d) := \texttt{ratedNC17}(d) \wedge \texttt{ageover17}(a)$$

Then, when downloading a specific trailer Alice also receive the policy related to that piece of data. For example, if *trailer* (the trailer that Alice download) is suitable for all the ages the action performed is as follows:

$$act_4 = \texttt{comm}(\mathit{Studio} \Rightarrow \mathit{Alice}, \varphi),$$

where

$$\varphi = \phi_{all}(\mathit{Alice}, \mathit{trailer}) \rightarrow \texttt{mayPlay}(\mathit{Alice}, \mathit{trailer})$$

The action of downloading is logged by Alice, and the log the state is updated as follows:

$$S_4 = S_3 \cdot \mathit{Alice} \mapsto \langle act_4, \emptyset, \emptyset \rangle$$

– Alice obtains the rating of the trailer, $\texttt{ratedAll}(\mathit{trailer})$, from Rating, which expresses that the trailer has no age restriction. Again this action is logged by Alice:

$$act_5 = \texttt{comm}(\mathit{Rating} \Rightarrow \mathit{Alice}, \texttt{ratedAll}(\mathit{trailer}))$$
$$S_5 = S_4 \cdot \mathit{Alice} \mapsto \langle act_5, \emptyset, \emptyset \rangle$$

– Alice watches the trailer on her device, which logs the action:

$$act_6 = \texttt{play}(\mathit{Alice}, \mathit{trailer})$$
$$S_6 = S_5 \cdot \mathit{Alice} \mapsto \langle act_6, \emptyset, \emptyset \rangle$$

– Alice obtains the video clip, *clip*, with the "pay per view" (1$) license from Studio. She also obtains the rating from the Rating Service (the trailer can seen by anyone who is over 13 years):

$$act_7 = \texttt{comm}(\mathit{Studio} \Rightarrow \mathit{Alice}, \varphi)$$

where,

$$\varphi = \phi_{age13}(\mathit{Alice}, \mathit{clip}) \rightarrow \big(!\texttt{pay}(\mathit{Alice}, 1\$) \rightarrow \texttt{mayPlay}(\mathit{Alice}, \mathit{clip})\big)$$
$$S_7 = S_6 \cdot \mathit{Alice} \mapsto \langle act_8, \emptyset, \emptyset \rangle$$

$$act_8 = \texttt{comm}(\mathit{Rating} \Rightarrow \mathit{Alice}, \texttt{ratedPG13}(\mathit{clip}))$$
$$S_8 = S_7 \cdot \mathit{Alice} \mapsto \langle act_7, \emptyset, \emptyset \rangle$$

– Next she pays for and plays the video clip. When Alice logs the play action, she also records that she is over 13 and that she has paid for this content.

$$act_9 = \texttt{pay}(\mathit{Alice}, 1\$)$$
$$S_9 = S_8 \cdot \mathit{Alice} \mapsto \langle act_9, \emptyset, \emptyset \rangle$$

$$act_{10} = \texttt{play}(\mathit{Alice}, \mathit{clip})$$
$$S_{10} = S_9 \cdot \mathit{Alice} \mapsto \left\langle \begin{array}{c} act_{10}, \\ \{\texttt{ageover13}(\mathit{Alice})\}, \\ \{!act_9\} \end{array} \right\rangle$$

The execution trace for this scenario is $act_1 \ldots act_{10}$. The trace of actions logged by Alice, viz. $S_{10}(\mathit{Alice})$, is $act_4 \ldots act_{10}$.

3.3 Accounting for Executed Actions

Agents may be audited by some auditing *authority*, at some point in the execution of the system. Intuitively, when some agent is about to be audited, an auditing authority is formed. This authority will audit the agent to find whether it is able to account for its actions. The knowledge of the auditing authority is represented by an 'evidence' trace $\mathcal{E}$ which is a sub-trace of the execution of the system (up till now). Which actions are in $\mathcal{E}$ depends on the power (and possibly the interests) of the authority; a more powerful authority will in general be able to collect a larger evidence trace.

Definition 5 (Accountability) We say that an agent A *correctly accounts* for an action act if it provides a valid proof of $\Gamma_1, \Gamma_2, \Delta \vdash_a pro(act, A)$ where Γ_2 is a list of actions from the log of A and Γ_1, Δ are the conditions and obligations[3] logged with the action if A logged this action or empty otherwise. The new *actions revealed* by the proof are the actions in Γ_2 and Δ which are not already in the evidence trace $\mathcal{E}$.

We say an agent A *passes the audit (or accountability test)* $\mathcal{E}$, written $ACC(A, \mathcal{E})$, if it correctly accounts for all actions in $\mathcal{E}$ and for all actions revealed by proofs it provides.

[3] The obligations are labeled actions rather than actions. Thus to be precise we should say Δ is the list obtained by removing the labels from the actions in the list of obligations.

An agent has to account for the actions in the evidence trace by providing a valid proof. If it logged the action, it can use the conditions and obligations it logged with the action in the proof. If it did not log the action it will have to provide a proof without any conditions or obligations. This shows why it is advantageous for agents to log actions.

In providing a proof, the agent may reveal actions that were not yet known to the auditing authority. These actions are added to the actions to be audited i.e. the evidence trace. Above we only consider a single agent but it is also possible to have an authority which iteratively audits all agents involved in actions in the evidence trace. In this case newly revealed actions may require the authority to revisit agents or add new agents to its list. (However, as the number of actions to be audited is always limited by the number of actions executed in the system we know the process will still terminate.)

Example 7 (Auditing) Let us now assume that an auditing authority is formed which audits Alice. The initial evidence trace $\mathcal{E}$ of the authority consists of $act_4 \dots act_{10}$.

The only actions which Alice has to justify, w.r.t. $\mathcal{E}$ are the actions act_6 and act_{10}; they are the only actions for which she has a proof obligation.

$$\begin{aligned} pro(act_6, Alice) &= pro(\texttt{play}(Alice, trailer), Alice) \\ &= \texttt{mayPlay}(Alice, trailer) \\ pro(act_{10}, Alice) &= pro(\texttt{play}(Alice, clip), Alice) \\ &= \texttt{mayPlay}(Alice, clip) \end{aligned}$$

Justification Proof. We show how Alice produces a justification proof for the logged action

$$\langle act_{10}, \{\texttt{ageover13}(Alice)\}, \{!act_9\}\rangle$$

Alice needs to prove the following policy:

$$\begin{aligned} pro(act_{10}, Alice) &= pro(\texttt{play}(Alice, clip), Alice) \\ &= \texttt{mayPlay}(Alice, clip). \end{aligned}$$

Then she uses a proof finder to find the proof. She can use the condition $\texttt{ageover13}(Alice)$ and the obligation $!act_9$. The proof is shown in Figure 4.

A justification proof for the logged action $\langle act_6, \emptyset, \emptyset\rangle$ can be provided in a similar way. It is reported in Figure 6

Honest Strategy A straightforward strategy for an honest agent A to be able to pass any audit is to, before executing an action act, derive the proof obligation $pro(act, a)$. If any obligation needs to be fulfilled, the agent fulfills it and logs the actions. If any condition or obligation needs to be fulfilled, then the action act itself is also logged.

Remark 2 (Accountability of honest agents) If agent A follows the *honest strategy*, then for any system execution and any auditing authority with evidence trace $\mathcal{E}$, we have that $ACC(A, \mathcal{E})$ holds.

4 Implementation

In this section we describe how we implemented the two principal components of our framework: the proof checker to be used by the auditors to check justification proofs and the proof finder to be used by the agents to find compliance proofs. But before we go into details about the implementations, we briefly describe how the proof finder and the proof checker interact.

4.1 Proof Checking and Proof Finding

Assume that an agent a has performed an action α and that the auditing authority wants a to justify. (See Figure 5.) First, (1) agent a is audited for action α at time t. Agent a now selects an excerpt ϵ of its log and a policy ϕ that is a's proof obligation for action α and (2) tries to find a proof of $\epsilon \vdash_a \phi$ with the proof finder. Then (3) the proof π and the excerpt ϵ are sent to the auditor for checking (4) and finally, (5) the auditor checks that π is indeed a proof of $\epsilon \vdash_a \phi$ by using the proof checker (6).

4.2 The Proof Checker

The audit logic allows authorities to check that compliance proofs are valid. To support this, we formalized the inference rules of the proof system, using the logical framework Twelf [19]. Twelf uses the *propositions-as-types* correspondence, also called the Curry-Howard isomorphism. Proof checking in Twelf thus reduces to type-checking. Earlier research in proof-carrying code has shown that Twelf uses a convenient notation for proofs to be sent and checked by a recipient [3,25,16]. This notation can be seen in Figure 6. The implementation of the inference rules in Twelf consists of about 100 lines of code.

Let us now return to the owns-L rule, in the proof system (Figure 3). It is cumbersome to define (in Twelf) the set of data $data(\phi)$ that the policy ϕ depends on. Set-theory would be required to define $data$ for compound policies. However ϕ in the owns-L rule can be restricted to atomic policies,

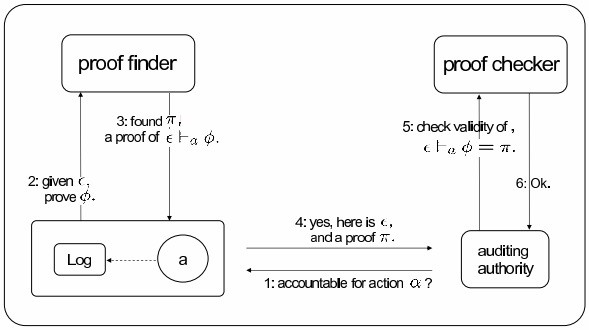

Fig. 5 The role of the tools in the event of an audit.

```
thm: (entail alice
    (cons (ageOver13 alice) nil)
     (cons (comm rating alice  (ratedPG13 clip)) (cons (comm studio alice
                   (imp  (and (ageOver13 alice) (ratedPG13 clip)) (!imp (pay alice bank)
                   (mayPlay alice clip)))) (cons (pay alice bank) nil)))
      (cons (pay alice bank) nil)
       (mayPlay alice clip))=
 (obs_act (obs_act (says_l (imp_l (and_r (perm_g1_2 init) (says_l init) append_nil)
           (!imp_l init) append_nil)) concl_comm) concl_comm).
```

Fig. 4 Alice's proof that she may play a clip provided she paid the bank.

provided we add the rule

$$\frac{\Gamma_1, \mathtt{says}(b, (\mathtt{owns}(a, d)), c); ; \vdash_a \mathtt{says}(b, \psi, c)}{\Gamma_1, \mathtt{owns}(a, d); ; \vdash_a \mathtt{says}(b, \psi, c)}\text{owns-says.}$$

It can be shown that the proof system obtained in this way is sound and complete with respect to the one in Figure 3. Soundness follows since, owns-says is provable using cut, the general form of owns-L and weakening. Completeness can be shown by proving (the general) owns-L, by case-analysis over ϕ. If ϕ is atomic, then the restricted form of owns-L is applicable. If ϕ is of the form $\mathtt{says}(b, \psi, c)$, then ϕ can be stripped using owns-says and refine. If ϕ is of another compound form, then ϕ can be stripped using other rules. In owns-says, the formula on the right side of the entailment relation $\vdash_a$ is restricted to satisfy the sub-formula property.

4.3 The Proof Finder

To respond to audits, an agent should be able to find compliance proofs based on its log. To do this in an automated manner, we implemented a proof finder (automatic theorem prover), using SWI-Prolog. While the proof checker is an implementation of the inference rules in Twelf, the proof finder consists of a representation of the inference rules in Prolog, together with some modules for the generation of the proof in a format appropriate for the proof checker. There are about 400 lines of Prolog code.

Cut Elimination For the proof system presented here, the cut rule is admissible, *i.e.,* if a policy is derivable using the cut rule then there is also a derivation of that policy without cut. Although *cut-admissibility* holds for the sequent calculus formalization of first-order logics[12], it is not trivial that it also holds for our logic, having introduced new logical rules to deal with *says to* and *owns*. The cut-elimination proof is included in the appendix. Cut-elimination has two important consequences: First, the *sub-formula property*[4] is satisfied, allowing for a more efficient proof search. Second, consistency of the logic is a consequence of cut-admissibility.[5]

[4] The formulas used in the premises are sub-formulas of those in the conclusion.

[5] Without the cut-rule, consistency normally follows, since there is no other rule that can introduce *falsity*. For the audit logic, it is easy to see that the formula $(\forall a, d.\ a\ \mathtt{owns}\ d)$ cannot be introduced without cut (except in some degenerated cases).

Prolog's resolution (backtracking) algorithm is used to perform proof search. In spite of cut-admissibility, the proof finder does not always terminate. The audit logic is an extension of (the fragment without disjunction and existential quantification of) predicate logic, which is in general undecidable, only certain fragments are decidable [11]. Note however that in our framework, since proof finding is only done by the agents, undecidability has no impact on the authority. In many other access control frameworks a decidable fragment of predicate logic is chosen, to prevent that undecidability complicates security decisions.

A sample proof output by the proof finder is reported in Figure 6. The proposition to be proven is written before the '=' sign. The proof is after the '=' sign.

To compare the different formalizations we show in Figure 7 how the says-L rule is written in the Twelf code and in the Prolog code. In the Prolog code, the second line in the says-L rule, is used to find a permutation of Γ_1 such that $a\ \mathtt{says}\ \phi\ \mathtt{to}\ b$ is on the first position. This replaces the need for separate permutation rules in the proof finder (see Figure 3), which would be inefficient. When such a permutation is found, then the context is permuted and permutation steps are printed in the proof for the proof checker. Because these permutation steps can become lengthy, we abbreviate using lemma's, that are available at the proof checker, i.e. perm_g1_2 is the lemma which takes the second element of Γ_1 and puts it in the first position. For the full source code of both tools, we refer to our online demo [1].

Note that the demonstrated proof finder is by no means a state-of-the-art theorem prover. But it shows a possible approach to implement policy-based access control. A future possibility may be to use lean theorem proving [5], which is particularly fast at solving simple problems but slower for complex logical problems.

5 Discussion

Having completed the description of the framework we now discuss questions relating to the practical applicability of the framework.

How expressive is the framework? We have presented a flexible framework with an expressive language containing a fine grained form of delegation. The framework also enables the use of use-once obligations thus allowing a permission to be 'consumed' when it it used. The policies in the languages,

```
thm: (entail alice
    (cons (ageOver13 alice) nil)
     (cons (comm studio alice (forall G304:tm object  (imp  (ratedAll
                                          G304) (mayPlay alice G304))))
       (cons (comm rating alice (ratedAll trailer)) nil))
      nil
       (mayPlay alice trailer))=
 (obs_act (obs_act (says_l (perm_g1_2 (says_l (forall_l (imp_l init init
    append_nil))))) concl_comm) concl_comm).
```

Fig. 6 A sample of the output of the proof finder.

$$\frac{\Gamma_1, \phi \vdash_a \psi}{\Gamma_1, \mathsf{says}(b, \phi, a) \vdash_a \psi}\text{says-L}$$

```
%% twelf: says_l rule
says_l: entail A (cons Phi Gamma1) Gamma2 Delta Psi ->
                            entail A (cons (says B Phi A) Gamma1) Gamma2 Delta Psi.

%prolog: says_l rule
entail(A,Gamma1,Gamma2,Delta,Psi,[Perms,' (says_l',Pf,')',Bras]):-
                            perm([says(_,Phi,A)],Tail,Gamma1,g1,Perms,Bras),
                            entail(A,[Phi|Tail],Gamma2,Delta,Psi,Pf).
```

Fig. 7 The says-L rule in formal notation, in Twelf code and in Prolog code.

however, can only be used to express permissions. It is not possible to express prohibitions, i.e. policies which forbid a certain action. Note that prohibitions are typically problematic in a distributed setting where messages can get lost, agents can be offline, etc. (See also comparison with other approaches in the next section and future work described in Section 7.)

For which application areas is the framework suitable? We have shown in the examples that the framework is applicable to a content protection scheme. As mentioned in the introduction, the framework also fits well the protection of private data: Privacy regulations typically require that the subject of the data can decide how data is to be used, i.e. decide the policy for the data. This requires a flexible, data oriented policy framework such as our audit logic. Also, privacy regulations often require protection of data but do not require that misuse of data is completely impossible, again fitting with our auditing approach where misuse is deterred rather than prevented. Although the framework is not applicable in areas where strict guarantees are needed it can be combined with access control mechanism as a method to check usage of data after access has been granted.

How realistic are the assumptions for the framework? For our auditing logic we make several assumption about the environment the framework is applied in. The most obvious requirement for the audit logic is that agents are auditable i.e. that agents can be held accountable for their actions. Additionally there must be an auditing authority that is authorized to do audits. To let the auditing be an effective deterrent against policy violation, the authority should be able to gain knowledge of and audit a significant portion of the executed actions. When we consider a scenario with cooperating companies these conditions are typically met. The auditing processes can simply be a formalization of the audit trails which are already common practice. In other settings such as open systems, it may be hard to hold agents accountable for their actions.

How realistic other assumptions are again depends on the scenario and on the actions being executed. For example, the requirement of secure logging is an abstraction of several techniques that would need to be applied. When an agent claims that an action has happened then the authority should be able to check this. For communication one needs to achieve unforgeability of messages; A simple signature is already sufficient to show that an agent was indeed given a policy by another agent, but if e.g. the time of sending is important or agents may be working together against an auditor then stronger mechanisms need to applied. The choice of implementation techniques needs to be matched with the requirements for the specific scenario.

We have assumed that delegation of responsibility is always possible; an agent can use all policies it receives. One may want to restrict this by requiring some trust in the agent that provides the credential. (See also the discussion in Section 7.)

How efficient is the framework? Although the audit logical is in theory undecidable, this is not necessarily an issue in practice [3]. When building proofs a user typically only needs to take a limited, and definitely finite set of actions, users and data items into account. Proof finding may still be difficult on resource constrained devices. Libraries of standard proofs for often used policies and actions could be used in such cases. The audits can also be performed efficiently; the auditor only needs to check a provided proof which is a simple task. To log an action, a few facts need to be recorded. It seems likely that this can be done using

a relatively small amount of storage. Also, the logging is completely distributed and agents only need to record actions that they observe and in which they are interested. Still, as the system runs the logs of agents grows. A method to remove actions from the log that have been correctly accounted for or actions that provide permissions that are no longer needed will likely be needed in a practical application.

How usable is the framework? Thanks to the automatic proof finding tool the user only needs to select the obligations to use and 'press the button' to obtain a proof. The policy language uses intuitive and easy to understand operators but a policy can quickly become hard to read when it grows. If, for example, a user needs to consent to a policy, this policy and its consequences should be clear. Thanks to its logical nature, the policy language is suitable for checking that a given policy satisfies certain requirements. For example, a user could check if the suggested policy is a refinement of its own preferred policies. Nevertheless, tools for readable presentation of policies such as available for well established systems, such as P3P for privacy, would aid in a more widespread applicability.

6 Related Work

The framework presented here describes a logic for policies combined with a-posteriori allowance checks of performed actions. Compared to access control, in our framework, access is always granted; only later it is determined whether the requestor had permission to do what it did.

There is a large body of literature on logics in Access Control (see the survey by Abadi [2]). Here, we mention some of the proposals. John DeTreville designed Binder [10], a logic-based security language based on Datalog. Binder includes a special predicate, *says*, used to quote other agents. Our *says* construct differs in two aspects from Binder's: First, ours includes a target agent (see Section 2.4); Second, we allow nested *says*, while, when communicating a policy (i.e. importing a clause in Binder), care must be taken to avoid nested *says*, since it may introduce difficulties in their setting.

BLF [25] is an implementation of a Proof-Carrying-Code framework that uses both Binder and Twelf, which however focuses on checking semantic code properties of programs.

Sandhu and Samarati [22] give an account of access control models and their applications. Bertino et al. [6] propose a framework for reasoning on access control models, in which authorization rules treat the core components Subjects, Objects and Privileges. Sandhu and Park [17] take a different approach with their UCON-model, in which the decision is modeled as a reference monitor that checks the three components: ACL, Conditions and Obligations. This inspired the separation we made in this paper. Obligations and conditions are also prominent in directives on privacy and terms of use in DRM. In the work of Samarati et.al. [20], a discussion about decentralized administration is presented. Specially, the revocation of authorizations is addressed. We do not address such issues in our framework.

In the privacy languages P3P and E-P3P [4], policies describe which actions will be performed on private data. There are some fundamental differences between the P3P framework and our audit logic. In P3P, policies are not created by the subjects of the data and sent. Instead policies are already in place and may even be unknown to the subjects. The concept of *purpose* of an action is used. E-P3P also uses negation of policies to meant that certain actions will not be performed, this requires special care to avoid problems in a distributed setting.

More related to our auditing by means of proofs, Appel and Felten [3] propose the Proof-Carrying Authentication framework (PCA), also implemented in Twelf. Differently from our work, PCA's language is based on a higher order logic that allows quantification over predicates. Their *says* predicate does not have a target (unlike ours) and it satisfies $\phi \Rightarrow a \textit{ says } \phi$ and $a \textit{ says } \phi \mathbin{\&} a \textit{ says } (\phi \rightarrow \psi) \Rightarrow a \textit{ says } \psi$. Their system is implemented as an access control system for web servers, while in our case we focus on a-posteriori auditing.

Another logic for access control is presented in a manuscript by Garg and Pfenning [13]. Their logic is constructive (like ours) and they prove that the cut-rule is admissible (like we do). They have a *says* construct with similiar properties as the *says* in [3]. Garg and Pfenning also introduce a concept of *non-interference* for access control policies, to characterize classes of policies (propositions) whose presence or absence has no effect on proofs of certain other policies.

In our framework, unlike [3] and [13] policies depend on agents' logs and we also consider conditional obligations.

7 Conclusions and Future Work

We have presented a flexible usage policy framework which enables expressing and reasoning about policies and user accountability. Enforcement of policies is difficult (if not impossible) in the highly distributed setting we are considering. Instead, we propose an auditing system with best-effort checking by an authority depending on the power of the authority to observe actions. A notion of agent accountability is introduced to express the proof obligation of an agent being audited. To the best of our knowledge, the framework presented here (an extension of our earlier work [7,9]) is the first to describe a logic for policies combined with a-posteriori compliance checks of performed actions.

Our proof system has been formalized using the proof checker Twelf and a proof finder has been implemented in Prolog. Agents can develop proofs using the proof finder and the proof checker allows an authority to check the agents' proofs.

Our obligations cover pre- and post-obligations but not yet ongoing obligations [21]. The setup does, with an adaption of the definitions of accountability, seem to provide the means to include this type of obligations. The obligations in

our framework are both 'use once', e.g. $!pay(\$10)$ and 'use as often as needed' $?pay(\$10)$.

In our system, we have a rule for delegation of policies. If an agent X *says* a policy ϕ to Alice, then Alice may use ϕ without any requirements and it is X's responsibility to show that it had the permission to say ϕ to Alice. However, Alice may only want to use a policy from X if she (i) *knows* X, (ii) *authenticates* X, and (iii) *trusts* X. All these issues are (intentionally) abstracted away in our approach, as they seem to be orthogonal to our aims. For example, in (iii), the required level of trust may depend on the policy provided by X or on the way Alice is going to use the policy. There, a distributed trust management system (e.g. [15]) could be employed to obtain the required level of trust.

References

1. AC^2 proof tools at http://www.cs.ru.nl/paw
2. Abadi, M.: Logic in access control. In: P.G. Kolaitis (ed.) Proc. of the 18th IEEE Symposium on Logic in Computer Science (LICS), pp. 228–233. IEEE Computer Society Press (2003)
3. Appel, A.W., Felten, E.W.: Proof-carrying authentication. In: G. Tsudik (ed.) Proc. of the 6th Conference on Computer and Communications Security (CCS), pp. 52–62. ACM Press (1999)
4. Ashley, P., Hada, S., Karjoth, G., Schunter, M.: E-p3p privacy policies and privacy authorization. In: P. Samarati (ed.) Proc. of the ACM workshop on Privacy in the Electronic Society (WPES 2002), pp. 103–109. ACM Press (2002)
5. Beckert, B., Posegga, J.: leantap: Lean tableau-based deduction. J. Autom. Reasoning **15**(3), 339–358 (1995)
6. Bertino, E., Catania, B., Ferrari, E., Perlasca, P.: A logical framework for reasoning about access control models. ACM Transactions on Information and System Security (TISSEC) pp. 71–127 (2003)
7. Cederquist, J.G., Corin, R.J., Dekker, M.A.C., Etalle, S., den Hartog, J.I.: An audit logic for accountability. In: A. Sahai, W.H. Winsborough (eds.) 6th Int. Workshop on Policies for Distributed Systems & Networks (POLICY), Stockholm, Sweden, pp. 34–43. IEEE Computer Society Press, Los Alamitos, California (2005)
8. Chong, C.N., Corin, R., Etalle, S., Hartel, P.H., Jonker, W., Law, Y.W.: LicenseScript: A novel digital rights language and its semantics. In: K. Ng, C. Busch, P. Nesi (eds.) 3rd Int. Conf. on Web Delivering of Music (WEDELMUSIC), pp. 122–129. IEEE Computer Society Press (2003). URL http://www.ub.utwente.nl/webdocs/ctit/1/000000bf.pdf
9. Corin, R., Etalle, S., den Hartog, J.I., Lenzini, G., Staicu, I.: A logic for auditing accountability in decentralized systems. In: T. Dimitrakos, F. Martinelli (eds.) Proc. of the 2nd IFIP Workshop on Formal Aspects in Security and Trust (FAST), vol. 173, pp. 187–202. Springer (2004)
10. DeTreville, J.: Binder, a logic-based security language. In: Proc. of the IEEE Symposium on Research in Security and Privacy (S&P), pp. 105–113. IEEE Computer Society Press (2002)
11. Dowek, G., Jiang, Y.: Eigenvariables, bracketing and the decidability of positive minimal intuitionistic logic. Electronic Notes in Theoretic Computer Science. **85**(7) (2003)
12. E. M. Szabo, editor: The Collected Papers of Gerhard Gentzen. North Holland (1969)
13. Garg, D., Pfenning, F.: Non-interference in constructive authorization logic (2006)
14. Jajodia, S., Samarati, P., Subrahmanian, V.S., Bertino, E.: A unified framework for enforcing multiple access control policies. In: J. Peckham (ed.) SIGMOD 1997, Proc. International Conference on Management of Data, pp. 474–485. ACM Press (1997)
15. Li, N., Mitchell, J., Winsborough, W.: Design of a role-based trust-management framework. In: M. Abadi, S.M. Bellovin (eds.) Proc. of the IEEE Symposium on Research in Security and Privacy (S&P), pp. 114–130. IEEE Computer Society Press (2002)
16. Necula, G.C.: Compiling with proofs. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1998)
17. Park, J., Sandhu, R.: Towards usage control models: Beyond traditional access control. In: E. Bertino (ed.) Proc. of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 57–64. ACM Press (2002)
18. Pfenning, F.: Linear logic course handouts. http://www.cs.cmu.edu/ fp/courses/linear.html (2002)
19. Pfenning, F., Schürmann, C.: System description: Twelf — A meta-logical framework for deductive systems. In: H. Ganzinger (ed.) Proc. of the 16th International Conference on Automated Deduction (CADE), pp. 202–206. Springer (1999)
20. Samarati, P., De Capitani di Vimercati, S.: Access control: policies, models, and mechanisms. In: R. Focardi, R. Gorrieri (eds.) Foundations of Security Analysis and Design (FOSAD), LNCS, vol. 2171, pp. 137–196. Springer (2001)
21. Sandhu, R., Park, J.: Usage control: A vision for next generation access control. In: V. Gorodetsky, L.J. Popyack, V.A. Skormin (eds.) 2nd International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security MMM-ACNS, *LNCS*, vol. 2776, pp. 17–31. Springer (2003)
22. Sandhu, R., Samarati, P.: Access control: Principles and practice. IEEE Communications Magazine **32**(9), 40–48 (1994)
23. W3C: A p3p preference exchange language 1.0 (appel1.0). www.w3.org/TR/P3P-preferences (2002)
24. Wang, X., Lao, G., De Martini, T., Reddy, H., Nguyen, M., Valenzuela, E.: XrML: eXtensible rights markup language. In: M. Kudo (ed.) Proc. 2002 ACM workshop on XML security (XMLSEC-02), pp. 71–79. ACM Press (2002)
25. Whitehead, N., Abadi, M., Necula, G.C.: By reason and authority: A system for authorization of proof-carrying code. In: 17th Computer Security Foundations Workshop, pp. 236–250. IEEE Computer Society Press (2004)

A Cut-elimination

In this section we proof a so called cut elimination theorem stating that the cut rule is actually redundant; anything proven using the cut rule can also be proven without using this rule. The cut-rule can be written as follows.

$$\frac{\Gamma \vdash_a \phi \qquad \Gamma, \phi \vdash_a \psi}{\Gamma \vdash_a \psi}\text{cut}$$

Here ϕ is called the cut-formula. Below we do not consider the left and right rules for obligations, as they do not interfere with the cut-elimination property. For readability, we only write the non-linear context Γ in the sequents in this section, as the other two contexts are irrelevant for this proof. For the same reason, in the sequel we ignore the left and right rules for $! \rightarrow$ and $? \rightarrow$.

The cut elimination can be phrased in words as: When we can proof some lemma (ϕ) and prove a formula ψ using this lemma then the formula ψ can also be proven directly. Not having this very intuitive property would indicate a very exotic logical system indeed. Cut-elimination theorems, due to Gentzen, are considered a central issue in field of logics. A cut-elimination theorem exists for first-order logic as well as for a number of other standard logical systems. The elimination of a cut rule often plays an important role in showing consistency and decidability of a logic.

Another reason, described in Section 4, for eliminating the cut rule is that the rule does not satisfy the so called sub-formula property; the cut-formula in the premise may be completely absent in the conclusion. The sub-formula property is important to be able to implement an efficient proof search. Thus for proof search one can simply restrict to the system without the cut rule.

A.1 The Proof

For proving the cut elimination theorem for our logic we follow a standard approach [18]: We show by induction that proofs including a cut rule can be transformed into proofs without this rule.

Our induction assumption states that, if we have a cut free proof $\mathcal{D}$ for $\Gamma \vdash \phi$ and a cut free proof $\mathcal{E}$ for $\Gamma, \phi \vdash \psi$ then we also have a cut free proof $\mathcal{F}$ for $\Gamma \vdash \psi$. This induction assumption is applied if the cut formula (ϕ) is simplified or if the cut formula stays the same and one of the proofs is shortened (and the other proof is not lengthened).

Note that any proof for $\Gamma \vdash \phi$ can be *weakened* to a proof for $\Gamma, \psi \vdash \phi$ by using the same rules but simply adding ψ in each step.

We distinguish the following cases, based on the last rule used in the proofs $\mathcal{D}$ and $\mathcal{E}$. Below, a formula is called principal in the rule, if the rule explicitly introduces the formula (either left or right of the $\vdash$). For clarity we show a table with the cases for $\mathcal{D}$ and $\mathcal{E}$. Here pr. denotes principal.

	$\mathcal{D}$ init	$\mathcal{D}$ owns-L	ϕ not pr. in $\mathcal{D}$	ϕ pr. in $\mathcal{D}$
$\mathcal{E}$ init	1	2	2	2
$\mathcal{E}$ owns-L	1	3	5	4
ϕ not pr. in $\mathcal{E}$	1	6	5	6
ϕ pr. in $\mathcal{E}$	1	3	5	7

The rules init and owns-L are the base-cases of the induction over the length of the derivation, so we treat them first. Please note that in the proof we ignore the rules concerning use once and use many obligations, $! \to R$, $! \to L$, $? \to R$ and $? \to L$, and the $\top R$ rule, which amount to trivial cases below.

1. $\mathcal{D}$ **ends in I.** When $\mathcal{D}$ consist of a single init (I) rule,

   $$\mathcal{D} : \frac{}{\Gamma', \phi \vdash \phi}\mathrm{I}$$

   (i.e. $\Gamma = \Gamma', \phi$) then applying contraction to $\Gamma', \phi, \phi \vdash \psi$, which is the conclusion of $\mathcal{E}$, gives us the required sequence $\Gamma', \phi \vdash \psi$. Thus $\mathcal{E}$ followed by contraction is a cut-free proof for this sequent.
2. $\mathcal{E}$ **ends in I.** When $\mathcal{E}$ consists of a single init (I) rule and ϕ is used,

   $$\mathcal{E} : \frac{}{\Gamma, \psi \vdash \psi}\mathrm{I}$$

   then the cut-formula is ψ and a cut-free derivation of ψ is simply $\mathcal{D}$. Otherwise, if ϕ is not used,

   $$\mathcal{E} : \frac{}{\Gamma', \psi, \phi \vdash \psi}\mathrm{I}$$

   (i.e. $\Gamma = \Gamma', \psi$), then a cut-free proof for the required sequent $\Gamma', \psi \vdash \psi$ is a single application of the init (I) rule.
3. $\mathcal{D}$ **ends in owns-L.** When $\mathcal{D}$ consists of a single application of the owns-L rule, then ϕ is atomic (see Section 4.2),

   $$\mathcal{D} : \frac{}{\Gamma', \mathsf{owns}(a, d) \vdash \phi}\text{owns-L}$$

   so if ϕ is principal in the last inference in $\mathcal{E}$ then this inference must use rule init (I), covered in 2, or owns_says or owns$_l$. In the latter two cases, one can simply contract the context to obtain the required sequent. In case ϕ is not principal in $\mathcal{E}$'s last step then we can apply the induction assumption for a smaller proof $\mathcal{E}$, see case 6.
4. $\mathcal{E}$ **ends in owns-L.** When $\mathcal{E}$ is owns-L and ϕ is used, then ϕ is an owns-predicate.

   $$\mathcal{E} : \frac{}{\Gamma, \mathsf{owns}(a, d) \vdash \psi}\text{owns-L}$$

   There are no cases for ϕ principal in $\mathcal{D}$'s last step except init and owns-L, both treated in the cases 1 and 3. In case ϕ is not principal in $\mathcal{D}$'s last step we can apply the induction assumption for a smaller proof $\mathcal{D}$, see case 5. Otherwise if ϕ is not used in owns-L,

   $$\mathcal{E} : \frac{}{\Gamma, \phi, \mathsf{owns}(a, d) \vdash \psi}\text{owns-L},$$

   then a cut-free derivation of ψ is a single application of the owns-L rule.
5. ϕ **is not principal in** $\mathcal{D}$**.** The cut-formula is not principal in the derivation $\mathcal{D}$ if the derivation ends in one of the (left) rules: $\to$L, $\forall$L, $\wedge$L$_1$, $\wedge$L$_2$, says-L, concl, owns_says. As an example we do the case for says-L.
   If the proof $\mathcal{D}$ consists of proof $\mathcal{D}_1$ followed by says-L:

   $$\mathcal{D} : \frac{\Gamma', \phi_1 \vdash \phi}{\Gamma', \mathsf{says}(b, \phi_1, c) \vdash \phi}\text{says-L} \qquad \mathcal{E} : \Gamma', \mathsf{says}(b, \phi_1, c), \phi \vdash \psi$$

   then by weakening $\mathcal{D}_1$ by adding $\mathsf{says}(b, \phi_1, c)$ and weakening $\mathcal{E}$ by adding ϕ_1 we get proofs for $\Gamma', \phi_1, \mathsf{says}(b, \phi_1, c) \vdash \phi$ and $\Gamma', \phi_1, \mathsf{says}(b, \phi_1, c), \phi \vdash \psi$ thus by induction (the weakened $\mathcal{D}_1$ is shorter than $\mathcal{D}$ and the weakened $\mathcal{E}$ is the same length as $\mathcal{E}$) there is a cut-free proof for $\Gamma', \phi_1, \mathsf{says}(b, \phi_1, c) \vdash \psi$ By applying says-L and then contraction we get to the required sequent $\Gamma', \mathsf{says}(b, \phi_1, c) \vdash \psi$.
   With the same trick of weakening, one proves the cases for the other left rules.
6. ϕ **is not principal in** $\mathcal{E}$**.** It is easy to see we can apply the induction assumption on the $\mathcal{D}$ and $\mathcal{E}_1$, which is $\mathcal{E}$ without the last step, to obtain a cut free proof $\mathcal{F}_1$ and then apply the same right rule as the righthand side of the sequents proven by $\mathcal{E}_1$ and $\mathcal{F}_1$ are the same.
7. ϕ **is principal in both** $\mathcal{D}$ **and** $\mathcal{E}$**.** Now there is one case left, i.e. where the cut-formula ϕ is principal in the last rule of $\mathcal{D}$ and $\mathcal{E}$. Here we split cases for different forms of the cut-formula and use the induction assumption for a sub-formula of the cut-formula.
   (a) **Subcase** $\phi = \phi_1 \to \phi_2$**.** There is one case for the last inference of $\mathcal{D}$:

   $$\mathcal{D} : \frac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash (\phi_1 \to \phi_2)} \to \mathrm{L}$$

   and $\mathcal{E}$'s last inference must be $\to$L:

   $$\mathcal{E} : \frac{\Gamma \vdash \phi_1 \qquad \Gamma, \phi_2 \vdash \psi}{\Gamma, (\phi_1 \to \phi_2) \vdash \psi} \to \mathrm{L}$$

   We can apply the induction assumption on the premise in $\mathcal{D}$ and the first premise in $\mathcal{E}$ to obtain a cut-free proof for $\Gamma \vdash \phi_2$ and again use the induction assumption on this proof and the second premise in $\mathcal{E}$ to obtain a cut-free proof the required sequent. (Both cases use a simpler cut formula.) The cases for ϕ with the connectives $\wedge$ and $\forall$ are done in the same way.
   (b) **Subcase** $\phi = \mathsf{says}(b, \phi_1, c)$**.** There is one case for the last inference in $\mathcal{D}$:

   $$\mathcal{D} : \frac{\Gamma' \vdash \phi_1}{\Gamma'', \mathsf{says}(b, \Gamma', c) \vdash \mathsf{says}(b, \phi_1, c)}\text{refine}$$

   Then $\mathcal{E}$'s last inference must be refine or says-L. In the case of says-L, c is the same as a so,

   $$\mathcal{E} : \frac{\Gamma'', \mathsf{says}(b, \Gamma', a), \phi_1 \vdash \psi}{\Gamma'', \mathsf{says}(b, \Gamma', a), \mathsf{says}(b, \phi_1, a) \vdash \psi}\text{says-L}$$

   By replacing the refine in $\mathcal{D}$ by a repeated application of says-L for each proposition in Γ', and then weakening with Γ'', one gets a (cut-free) proof of $\Gamma'', \mathsf{says}(b, \Gamma', a) \vdash \phi_1$. Now the cut-rule can be used to derive ψ, and the cut-formula is smaller hence by the induction assumption there is a cut-free derivation of $\Gamma'', \mathsf{says}(b, \Gamma', a) \vdash \psi$.
   On the other hand, when $\mathcal{E}$ ends in refine,

   $$\frac{\Gamma', \phi_1 \vdash \psi_1}{\Gamma'', \mathsf{says}(b, \Gamma', c), \mathsf{says}(b, \phi_1, c) \vdash \mathsf{says}(b, \psi_1, c)}\text{refine}$$

   then the induction assumption can be applied for the proofs $\mathcal{D}_1$ and $\mathcal{E}_1$ of the premises to reach the required sequent without the use of either refine-rule.
   (c) **Subcase** ϕ **is atomic.** There are two cases for the last step in $\mathcal{D}$ (where ϕ is principal), being init and owns$_l$, treated in the cases 1 and 3.

This completes the proof. □